Using The 9P Protocol In High-Performance Computing

Latchesar Ionkov
Andrey Mirtchovski
Los Alamos National Laboratory[*]
{lionkov,andrey}@lanl.gov

ABSTRACT

The 9P protocol, with its greatly reduced complexity, allows many different types of resources to be accessed as files over several different types of connections. It allows programmers to implement interfaces, which enable parts of a cluster to communicate as if they were connected directly. In this paper we describe the use of the 9P protocol as the main communication protocol between various components in heterogeneous clusters. We discuss the properties of 9P that allow us to write small but efficient communication libraries and to optimize them for the different transports available on the cluster. We describe several such transports and discuss algorithms, specific implementation details and performance results.

1. Introduction

Today's clusters are evolving from simple commodity compute nodes connected together via a standard network, to nodes comprising of a multitude of CPUs, computational accelerators and graphics processing units. The computational units in these clusters have different architectures (CPUs, GPUs, Cell Broadband Engine [5] [10] processors) and are using different interconnects (Ethernet, Infiniband, PCI Express, Element Interconnect Bus). The clusters are no longer homogeneous where a computational node is connected to every other node via a single interconnect.

This amalgam of different communication protocols increases the complexity of the software stack that will be run on a heterogeneous cluster. The most popular approach for current systems software is to hide the interconnect/computational differences using specialized libraries. For example, a distributed program consists of multiple programs running on the central processing unit and using Ethernet/Infiniband for communication. The programs themselves use libraries to offload some of their computations to GPU/Cell BE/FPGA units using PCI Express. In the case of the Cell Broadband Engine, the computation is further offloaded to its Synergistic Computational Units.

The use of heterogeneous interconnects and computational units makes it difficult to create cluster management software that allows effective control and monitoring because of the many different components that are involved. In order to reduce complexity, we are building a framework that allows us to speak with any computational component on a heterogeneous cluster via a simple file-based interface. This approach allows hiding the particularities of the interconnect topology when it is not important and yet expressing it, as well as the capabilities of the computational nodes, in the hierarchy of files when required. Our libraries fit in all areas of the cluster, from the end-user desktop to the Cell computational accelerator. All protocol communications are optimized for the various underlying transports with all access to the cluster's components occurring via POSIX-like file operations such as `open`, `read` and `write`.

2. Examples of Heterogeneous Cluster Architectures

The trend towards heterogeneous clusters has been well established in today's HPC environments. Organizations such as Los Alamos National Laboratory are building clusters to serve

*LANL publication: LA-UR-07-7865

different needs of various fields of science. In order to achieve high performance, these clusters must necessarily utilize hardware technology best suited for the particular task. In fact, we are seeing more and more dedicated solutions provided by vendors such as IBM to serve a particular science task (bioinformatics and biotechnology, for example) which are not usable for general computing needs. Moreover, the next generation of standard "homogeneous" clusters will most likely include several different accelerators which do not fit the general mode of operation of the CPU, such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) and others, which work in conjunction with the CPU to accelerate specialized computations.

Naturally, LANL, an organization which prides itself on being on the cutting edge of science, is taking steps to discover new, faster methods for performing scientific computation. Below we discuss several efforts to bring heterogeneity to high-performance computing: the Usable Supercomputer, the RoadRunner cluster and the yet unnamed cluster built at the Advanced Computing Lab at LANL.

2.1. Usable Supercomputer

The term Usable Supercomputer denotes an attempt by the computer science division of Los Alamos National Laboratory to simplify and reduce the efforts required to port existing code between successive generations of supercomputers.

The goal of High Performance Computing is to extract the maximum performance from the currently fastest hardware. In order to do that, more often than not the solution is to optimize code that will be running for a particular architecture and couple it very tightly with the hardware. Such tasks are not trivial and require great amount of knowledge of both the code and the hardware it will be running on. In some cases as much as half of the operational time of a supercomputer or an HPC cluster has been spent tinkering and optimizing code that will be run on it. Indeed, once a particular application has achieved what is considered maximum performance for a particular architecture, it is very compelling to continue employing that architecture even after new, faster hardware becomes available, thus saving porting effort and time.

In an effort to reduce the time between porting applications to the next fastest hardware, the Usable Supercomputer will attempt to create a framework that allows applications to be moved from one hardware configuration to another with minimal rewrite. In essence, Usable Supercomputer does not apply to heterogeneous clusters (although the next section will discuss their inevitability), but rather attempts to deal with the generational differences between successive HPC architectures.

Our effort in Usable Supercomputing will be at the systems communication level. We will present the 9P protocol [6] as a viable solution to the problem of micro-optimizing systems software communications to work on a particular interconnect. By moving system-related communication over to 9P we are free to optimize for the underlying interconnect without having to modify the protocol itself with each new architecture. The introduction of 9P does decrease performance by increasing communication overhead versus direct communication optimized for the interconnect (our measurements suggest 10 to 15 percent for the Cell's DMA, for example) we are confident that overall there will be an increase in productivity.

2.2. RoadRunner

RoadRunner is the next generation supercomputing cluster currently being built at LANL. It is the biggest system in existence to employ both general processing units (AMD Opteron nodes) as well as computational accelerators (IBM's Cell Broadband Engine [4]). RoadRunner can be considered the first fully heterogeneous cluster: it utilizes four different interconnects, three different types of processing units, two byte orders, two different operating systems and at least two different memory models. Below we list some of the components for RoadRunner:

CPU Types:

- 64-bit, x86 descendant AMD Opteron; also usable in 32-bit legacy mode for running 32-bit applications
- 64-bit Power PC processing unit for the PPU of the Cell Broadband Engine
- 128-bit Synergistic Processing Unit for the Cell SPE, which is expected to bear the brunt

of the computations for RoadRunner

Interconnects:

- Cell DMA Engine, facilitating communication between the PPU and the SPEs of the Cell
- PCI-e bus between the Cell and the Opteron's motherboard
- Infiniband network between the Opterons
- Gigabit Ethernet or 10-GigE for connecting between the cluster's head nodes and desktops

Operating Systems:

- Opteron nodes run a 64-bit version of the Linux operating system for the x86 architecture (aka x86_64)
- Cell BE PPU nodes run a 64-bit PowerPC version of the Linux operating system (aka PPC). It is unclear as of this writing whether the two OSs can be synchronized
- Cell BE SPE units can not run an operating system. Instead, software running on them must be initialized by the corresponding PPU for that Cell blade

There are several possible ways of writing and deploying code for this cluster, all of them stemming from the fact that its architecture is heterogeneous and currently no software exists that can successfully combine all of RoadRunner's components into a single application. All of the methods listed below, excluding the most basic one, running it as a legacy homogeneous cluster, will require major rewrites of existing code:

- Legacy Homogeneous Cluster: using only the Opteron parts, code written for older clusters could simply run on RoadRunner without modification, utilizing perhaps an MPI library for intra-node communication over Infiniband. In this form the accelerator components of each Opteron node, the Cell blades are unused.
- A cluster of accelerators: programs can use only the Cell BE accelerators to perform computation, using the Opterons as a staging environment. This involves a significant effort in designing communication libraries that can connect Cells over the Infiniband interconnect with minimal interference from the Opterons
- A mixed Opteron-Cell cluster: using the strengths of both the Opteron CPU and the Cell Broadband Engine, programs can employ them together for different parts of the computation cycle. This will involve an even greater effort in designing and implementing libraries that deal with bit ordering, communication over different interconnects and rewriting existing science applications to use the new libraries. On the systems side of things, resource administrating software will need to be optimized for speedy delivery of information and data in order to keep up with the computation. This arrangement, however difficult it is to achieve, has the biggest potential for increasing the cluster's performance closer to the theoretical limit for RoadRunner.

2.3. Other efforts

We are building a yet-unnamed cluster at the Advanced Computing Lab at LANL. The cluster will serve as a testbed for various groups within the lab and will attempt to expose programmers to a fully heterogeneous environment containing even more different hardware components than RoadRunner. The cluster comprises of 32 nodes each equipped with one or more multi-core CPUs, with each node connected to one or more Graphics Processing Units (GPUs) and FPGA boards. While the authors doubt that this cluster will enable the emergence of the sort of super-accelerated applications able to use FPGAs, GPUs, CPUs and other, yet uninvented

accelerators concurrently, we are nonetheless convinced that this cluster is a step forward in understanding and enabling synergies in a heterogeneous computational environment.

The variety and increase in different hardware architectures in heterogeneous environments pose a significant problem for those willing to employ them: both from the systems side (making them run) as well as the applications side (making them crunch numbers fast). As the Systems Software team at LANL, we are tasked with developing software which maintains a sound environment for job starting, data movement, monitoring and control of applications. The following sections will discuss our attempt to find a unified framework which works in a heterogeneous environment without overly hampering performance.

3. Representing System Resources as Files

In time honored tradition we have chosen the path to unifying the cluster control and information devices to follow that of Plan 9 [11], i.e., all resources to be presented and accessed as files. In doing this we have simplified greatly the creation and access to interfaces across the entire heterogeneous cluster, throughout its entire operation. We are using (or are planning to use) 9P to boot the cluster, to access it for job start-up over Ethernet and Infiniband, in computing libraries of the Cell SPEs and over the PCI Express bus for communication between the Opterons and the Cell processors.

Some of our implementations include:

3.1. Xbootfs

Xbootfs is a scalable 9P based protocol for distributing boot images in a cluster. The boot server serves the boot images (eventually different for the different architectures). The scalability is implemented by using the clients as temporary, second-tier boot servers. When a client connects to the boot server and gets its image via 9P, instead of continuing the boot process, it registers itself with the server as secondary boot server. When another client connects to the server, instead of serving the file directly, the server redirects it to one of the second-tier servers. After serving its time (usually 5 seconds) the secondary server unregisters itself and continues the boot process. Xbootfs was written by Ron Minnich and Li-Ta Lo.

3.2. XCPU

XCPU [8] uses a file system hierarchy of directories and files to export functionality from the compute node of a cluster in a way that allow it to be imported and viewed locally from one or more head nodes. XCPU provides a file interface for creating new jobs, fast distribution of the files necessary for the jobs work, control and monitoring of the job progress. XCPU supports heterogeneous clusters allowing the compute nodes and/or the head node to be of different architectures.

XCPU allows administrators to be very flexible on how the compute nodes will be configured. It allows splitting a single cluster into multiple small logical clusters, or merging multiple clusters into a single logical one.

XCPU nodes are mountable on the local Linux machine via the V9FS [2] [9] kernel module, allowing job starting and monitoring to be performed via standard UNIX shell commands and tools such as `cp`, `cat`, `ls` and `tail`. Figure 1 shows a possible configuration of several clusters mounted in a grid-like environment on a single node.

3.3. CellFS

The Cell Broadband Engine [10] is a new architecture aimed at providing high-performance computational facilities for scientific and media applications. Even though the Cell BE was designed initially for the gaming industry and specifically for Sony's PlayStation 3 game console, Cell computers have been embraced by the scientific community for their potential to deliver high-performance throughput to certain applications.

The Cell achieves its performance by utilizing two different types of computational units: a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPE) [3]. The operating system and user programs run on the PPE, while the 8 SPEs are used to offload specific computations.

The SPE implements a SIMD instruction set that is optimized for computationally intensive applications. It has 128 128-bit registers and 256 kilobytes of local store. The local store of the SPE is the only memory accessible directly by the load and store operations of the SPE

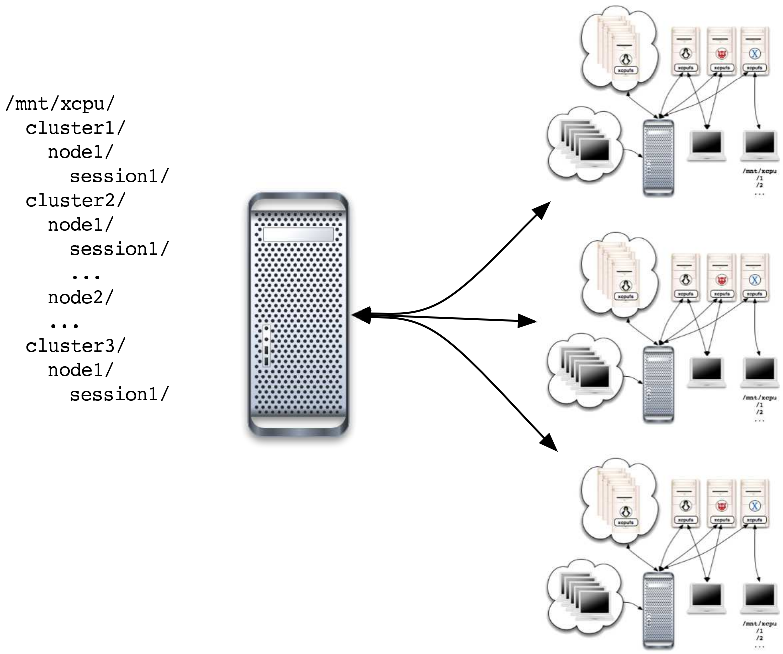


Figure 1: A constellation of XCPU-enabled clusters mounted on a single head node

instruction set. The size of the local store is further limited by the fact that it also stores the code of the currently executing program. This is the main limitation of the architecture: every access to main memory on the Cell needs to be explicitly scheduled. Programmers need to balance between how much code can be put on the SPE and how much data this code will have to work with.

The SPE uses asynchronous, coherent direct memory access (DMA) transfers to access the main memory, with memory address translation controlled by the PPE. There is support for up to 16 outstanding DMA requests on each SPE. DMA is programmed either by instructions executing on the SPE, by preparing DMA lists, or by inserting commands in the DMA queue from another processor (usually the PPE).

CellFS presents the CellBE resources (main memory, file system, etc.) as files. If a program running on the SPE needs to read data from the main memory, instead of issuing a DMA request, it reads from a file that represents the memory region. If it needs to send data to the PPE, or another SPE, it uses a pipe file. The model is flexible enough to allow access from the SPE to memory and other resources imported by the PPE from other nodes. For example the SPE can transparently read/write to the main memory of an Opteron node, or use the XCPU file interface to launch a job on another Cell processor.

3.3.1. CellFS Devices

Depending on the underlying storage we define five filesystems (listed in table 1) that are served by the PPE. Contingent on the prefix used with the full file name, different backing stores will be used to read or write the data associated with the file.

Most commonly used ones are #r and #U.

Code running on the SPEs accesses this file system via library calls corresponding to normal POSIX file operations. The following code opens a file named `test` in directory `/tmp` on the main file system of the Cell (presumably associated with a partition on the Cell's hard drive) and writes "num" bytes of "data" to it starting at offset 0:

```
fd = spc_open("#U/tmp/test");
```

Name and type	Description
#r	File server representing areas in main memory as files
#U	File server allowing operation on files existing on the UNIX file system accessible by the PPE. Files served by #U are mmap()-ed to main memory to increase I/O bandwidth
#R	Similar to #U, but changes to the files are not propagated to the disk. This is equivalent to a read-only file system, however it allows the SPEs to communicate data between each other as the computation progresses
#p	Clients can use this file system to create a named pipe which can be used to communicate between clients running on different SPEs.
#l	Log file system used by lightweight library routines replacing `printf()`

Table 1: File systems served by the PPE

```
spc_write(fd, data, num);
spc_close(fd);
```

Our model is easily extensible to provide access to more resources if necessary. Our tests show that we can achieve reasonably good performance for access to main memory of the Cell. This approach combines simplicity, very important when memory is limited, with extensibility.

3.3.2. Asynchronous execution

In order to use the SPE effectively, data transfer must not block the flow of the computation, i.e., the execution of code and I/O must overlap. There are two approaches for interface design that achieve the requirement: asynchronous I/O and multiple executions threads. The asynchronous I/O model has been experimented with in the double- and triple-buffered programming modes, but for a complete solution an implementation of an interface similar to POSIX Asynchronous I/O [7] must exist. We decided to implement a simple coroutine model that allows more than one function to run independently on the SPE since we believe that we have found a model simpler to implement and understand than asynchronous I/O. Our model does not require locking of information and provides completely deterministic execution even in the presence of multiple coroutines with overlapping I/O.

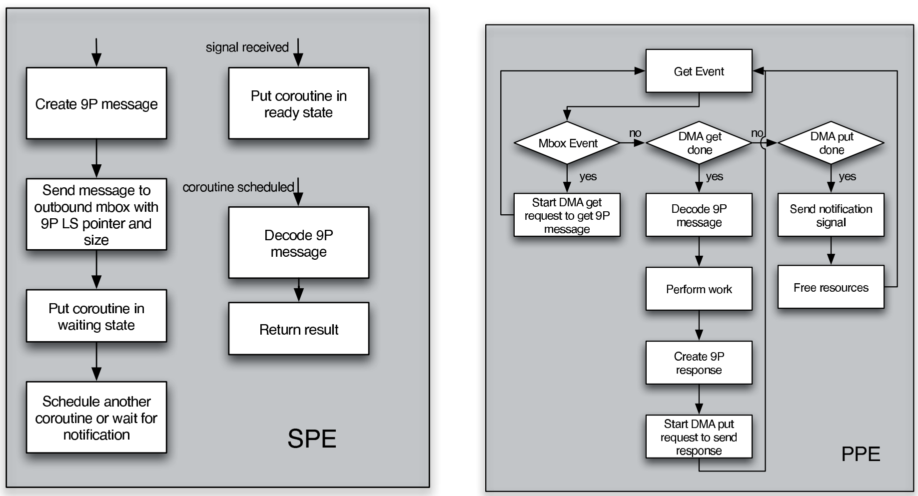
Fully deterministic coroutine execution is accomplished by allowing context switches to occur only on predefined places in the program's code. Whenever the running function performs an I/O operation, the library initiates the transfer to or from the PPE and passes the control to a different coroutine running on the same SPE. In order to keep both the interface and implementation simple, the switch between the coroutines can occur only when a function from the I/O interface is called. The fact that the switching points are known in advance requires no locking of data and greatly simplifies programming for the developers. We have borrowed this threaded model from Plan 9, where coroutines have been used successfully to implement various heavily used multi-threaded applications.

Coroutines are created similarly to POSIX threads. The `mkcor()` function receives a function pointer, parameter pointer and stack buffer pointers and handles the set-up stage for the new coroutine. A new coroutine is not immediately scheduled, but is put in a FIFO queue. The following code creates a coroutine with a stack size of 4096 bytes which prints a greeting.

```
char stack[4096];
void
cor(void *arg)
{
    spc_log("hello world\n");
}
```

```
void
cormain()
{
	mkcor(cor, NULL, stack, sizeof stack);
}
```

Figure 2(a) provides an example of the coroutine scheduling that may occur on an SPE with two coroutines.



(a) SPE Coroutine Scheduling

(b) PPE Workflow

There are several benefits of the coroutine model. Unlike threads or normally-scheduled OS processes, coroutines execute completely deterministically, thus removing the need for locking or mutual exclusion. The coroutines can share all variables defined in the SPE code as long as they do not try sharing the memory areas for which DMA may be in progress.

A disadvantage of the coroutine model is that it requires separate state to be kept and stored on the SPE for each coroutine.

4. 9P Transports

4.1. Element Interconnect Bus

Element Interconnect Bus (EIB) connects all elements within the Cell BE. It allows the elements to issue DMA requests, or send messages over message boxes. We use DMA requests and messages to implement the 9P transport. The client (running on the SPE) prepares the 9P message in its local store and sends a 32 bit message with its address to the PPE. The PPE issues a DMA request to transfer the message from the SPE local store to the main memory. The PPE then decodes the messages and executes the requested operation. Then a response is ready, the PPE issues a DMA request to transfer it to the same address in the SPE's local store and notifies the SPE via a signal.

4.2. Ethernet

We use the standard TCP/IP transport over Ethernet.

4.3. Infiniband

We use the standard TCP/IP transport over Ethernet. Also we are planning to implement an optimized transport using verbs directly.

4.4. PCI Express

Some Cell BE boards can be connected to a computer via PCI Express bus. The Cell BE boards have a DMA engine that allows operations similar to the one the EIB has. The implementation of the 9P transport over PCI Express to the Cell boards is a work in progress.

5. Tying it all together

Representing the computational nodes resources as files gives the cluster designers greater flexibility on how to combine and expose the cluster resources to the users. Exporting a XCPU tree to the SPU programs can give them the ability to start jobs on Opteron nodes over Infiniband. Or start computation on a GPU. For heterogeneous clusters it allows the designers to provide different views of the cluster that are independent of the actual interconnect topology. A heterogeneous cluster comprised of Cell BE nodes connected via PCIe to Opteron nodes which are connected via Infiniband can be presented as a Cell BE only cluster, as an Opteron cluster or as a heterogeneous cluster. The users that have Cell BE only application can use the Cell BE only view and their code doesn't have to be aware of the PCIe-to-Opteron routing.

The ultimate goal of our work is to allow the end user or application to interface to all components of a heterogeneous cluster that they need to in order to accomplish their work, without having to deal with the actual hardware components and interconnects involved. In essence, an end user or application should be able to "turn on" and "tune in" to only those parts of the cluster that they're interested in, circumventing the underlying hardware. Our implementations of 9P over the various interconnects of RoadRunner and the corresponding libraries and servers needed to access them allow programmers to employ a gigantic (by today's standards) machine such as RoadRunner in all the possible modes that we listed in section 2. A cluster of a thousand Opteron nodes, or a cluster of 2000 Cell processors should look identical to an end application.

6. Performance implications from moving to 9P

We have evaluated the performance of several of our components and have found that the slowdown compared to industry leading software is sufficiently small to warrant choosing our framework over more complex, albeit better optimized code.

6.1. XCPU

Our test benchmark, the code that we always attempt to outperform, is B-Proc [1]. There are two reasons for this: one is that as far as we know, B-Proc is the fastest cluster management system running on production clusters [12], and secondly, B-Proc is deployed on over ten thousand compute nodes on the Los Alamos National Laboratory clusters. If we want to create a replacement for B-Proc we must deliver similar, if not higher performance.

The results published in tables 2 and 3 are from Xcpu runs on SparkPlug, a B-Proc cluster consisting of 32 dual Opteron nodes with 1.8GHz processors, connected via Myrinet. Results summarized in table 2 are from runs performed on Blue Steel, a 256-node Opteron cluster connected with Infiniband. Of the 256 nodes we were allowed to run on 245. Both clusters are currently running B-Proc, with the ability to enable XCPU concurrently for testing. The times reported here are averaged from 1000 timed executions. We varied the number of nodes connected to and the size of the binary copied to the remote node.

# Binary Size	B-Proc (seconds)	XCPU (seconds)
64KB	3.1	2.5
1MB	4.6	7.3
8MB	5.0	13.5
16MB	5.5	20.3

Table 2: Tree-spawn results for B-Proc and XCPU on 245 Infiniband-connected nodes

We interpret these results to indicate that the Xcpu model scales well, even if it is not as fast as the predecessor it replaces. Our start-up algorithm is based on a tree-spawn method which offloads the bulk of the workload from the lead node onto the compute nodes, thus reducing the load on both the network and the file system up front.

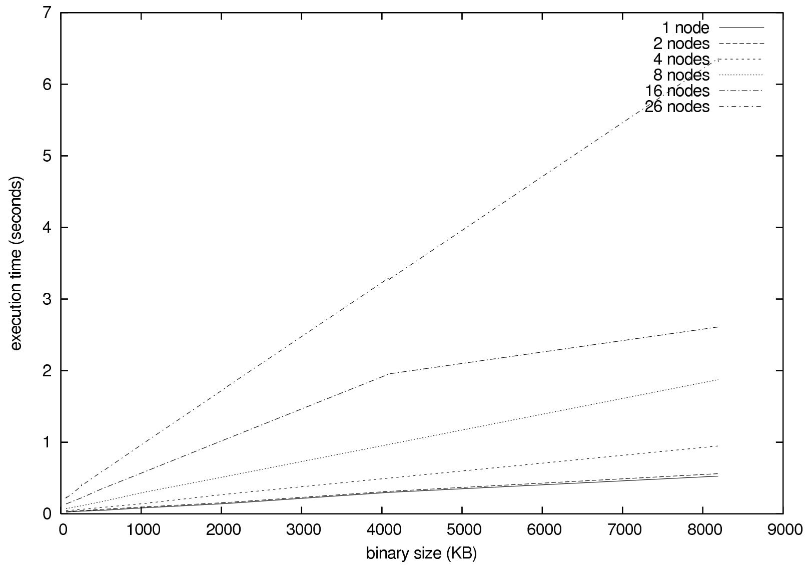


Figure 2: Execution time for a single binary execution on varied number of nodes

6.2. CellFS

In order to evaluate the performance of the proposed programming model, we created two benchmarks. We run them on a Cell blade system that contained two 3.2GHz Cell processors.

The first benchmark tests the maximum memory bandwidth that can be achieved. The SPE program opens a 128MB file stored on the Unix file system, then creates a number of coroutines, each of which reads and writes 16777216 8K blocks from the files. As mentioned in the previous section, the regular Unix files are mmap-ed when accessed from the SPE, so after the initial page faults that bring the file content to the memory, reading from the file is equivalent to accessing the main memory. We tested the program running on different number of SPEs and coroutines.

Table 3 shows the results of running the memory bandwidth benchmark. The memory bandwidth we achieve is much lower than the theoretical maximum. We believe that the main reason for that is that we don't use HugeTLB.

# SPE	DBL	# Coroutines						
		1	2	3	4	5	6	7
1	8.19	5.68	10.63	10.71	10.75	10.81	10.78	10.79
2	15.59	11.03	20.48	20.74	20.79	20.88	20.89	20.96
3	20.05	14.75	23.20	23.26	23.27	23.28	23.29	23.28
4	20.48	17.84	23.28	23.20	23.32	21.84	23.32	23.32
5	21.63	20.71	23.26	23.27	23.28	23.28	23.28	23.29
6	21.82	21.89	23.26	23.25	23.26	23.26	23.26	23.26
7	23.05	21.05	23.24	23.23	22.05	23.23	23.24	23.23
8	22.34	21.43	23.22	23.20	23.21	23.21	23.21	23.21

Table 3: Memory Bandwidth (in GB/s) for various numbers of active SPEs and Coroutines, as well as the benchmark double-buffered code

The second benchmark is a modification of the IBM optimized matrix multiplication workload (`src/workloads/matrix_mul` directory in Cell SDK 1.1). We left the optimized functions MatInit_MxM and MatMul_MxM intact, changing only the logic that reads and writes the blocks. Instead of implementing double-buffering model by issuing DMA transfers directly, our implementation uses two coroutines that each calculates half of the blocks assigned to the

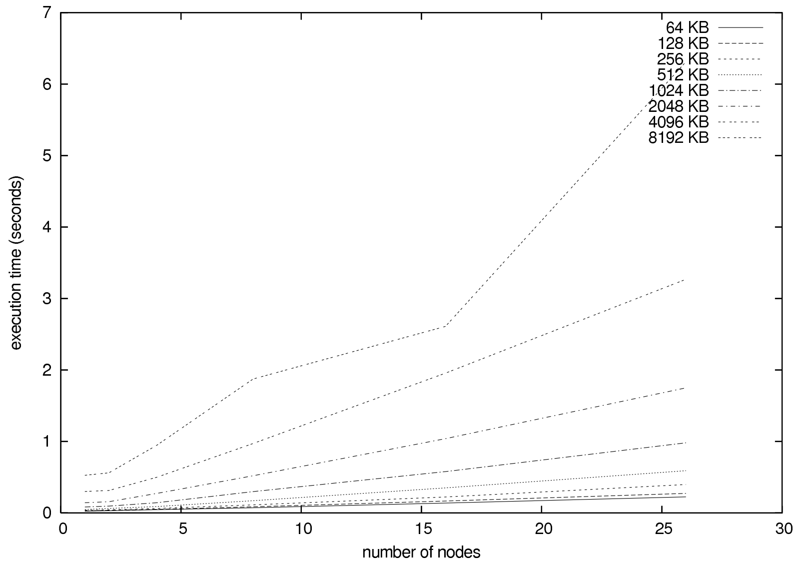


Figure 3: Execution time for various binary sizes

SPE. The initial data of the A and B matrices is stored in regular Unix files and the result of the multiplication is also stored in a regular file. In order to have fair comparison we modified the original matrix multiplication program to read the content of A and B from the same files instead of generating it on the fly.

The main function opens the files containing the matrices data, then creates two coroutines that do the actual multiplication:

```
static void
mmulcor(void *a)
{
    int i, j, k, blkid;
    char *ename;
    Ctx *ctx;

    ctx = a;
    for(blkid = ctx->blkfirst; blkid < ctx->blklast; blkid++) {
        i = (blkid >> shift) & mask;
        j = (blkid) & mask;

        block_read(ctx, i, j, 0);
        MatInit_MxM(ctx->c, ctx->a, ctx->b);
        for(k = 1; k < (N/M); k++) {
            block_read(ctx, i, j, k);
            MatMult_MxM(ctx->c, ctx->a, ctx->b);
        }

        block_write(ctx, i, j);
    }
}

static int
block_read(Ctx *ctx, unsigned int by, unsigned int bx, unsigned int idx)
{
    int sz;
    u64 offa, offb;
```

```
    offa = 4*(by*M*N + idx*M*M);
    offb = 4*(bx*M*M + idx*M*N);
    sz = sizeof(float) * M * M;

    spc_pread(afd, (u8 *) ctx->a, sz, offa);
    spc_pread(bfd, (u8 *) ctx->b, sz, offb);
    return 0;
}

static int
block_write(Ctx *ctx, unsigned int by, unsigned int bx)
{
    int sz;
    u64 offc;

    sz = sizeof(float) * M * M;
    offc = 4 * (by*M*N + bx*M*M);
    spc_pwrite(cfd, (u8 *) ctx->c, sz, offc);
    return 0;
}
```

Table 4 shows the results of running the original and the modified applications for tables of the specified size. The results indicate that the slowdown incurred by using our libraries is between 10% and 16% for medium-sized matrices. Figure 4 plots the slowdown as a function of the matrix size and the number of SPEs used in the computation.

# SPE	256x256		512x512		1024x1024	
	CellFS	Std.	CellFS	Std.	CellFS	Std.
1	17.65	15.56	141.43	124.22	1131.98	993.00
2	8.85	7.81	70.75	62.16	566.38	496.71
4	4.45	3.91	35.46	31.09	283.36	248.40
8	2.28	1.98	17.88	15.57	142.54	124.29

Table 4: Time to run 10000 multiplications of two single floating point matrices of the specified size

7. Conclusions

We have presented our view for the future deployment of the 9P protocol in High Performance Heterogeneous computing at the Los Alamos National Laboratory. We expressed our hope, that by using a single, standard protocol, we can reduce the complexity of interconnecting and interfacing between the various components and accelerators arriving with the next-generation HPC clusters.

We presented performance results for various of our framework components and expressed satisfaction with the fact, that replacing complex, highly optimized libraries with a simpler, easy-to-write-for file-based interface does not impact performance unnecessarily badly.

References

[1] Sung-Eun Choi, Erik A. Hendriks, Ronald G. Minnich, and Matthew J. Sottile. Life with ed - a case study of a linuxbios/bproc cluster.

[2] Latchesar Ionkov Eric Van Hensbergen. The v9fs project. http://v9fs.sourceforge.net.

[3] B. Flachs, S. Asano, S. H. Dhong, H. P Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. The microarchitecture of the streaming processor for a cell processor. In *IEEE International Solid-State Circuits Conference*, pages 184–185, 2 2005.

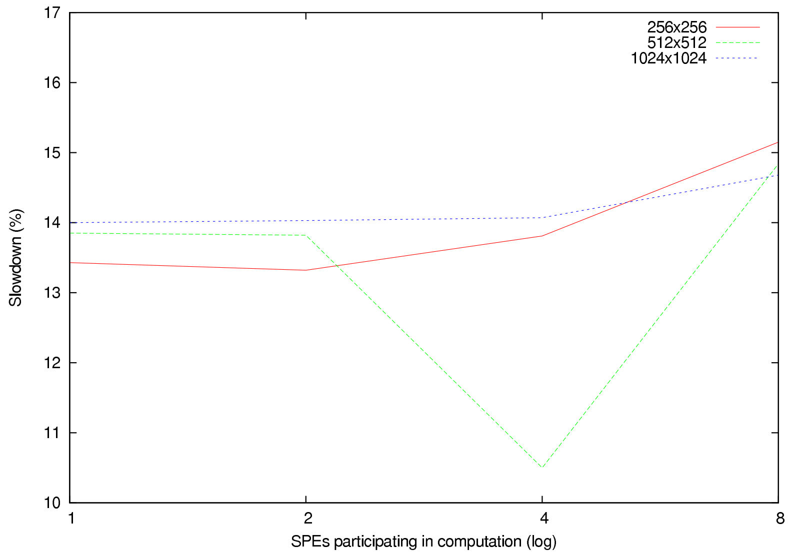


Figure 4: CellFS Slowdown vs Standard Double-Buffered Algorithm for Matrix Multiplication

[4] IBM. Cell broadband engine architecture, version 1.0, 2005.

[5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Mauerer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. & Dev.*, 49:589–604, 2005.

[6] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[7] The POSIX Asynchronous I/O Library Manual. At http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[8] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[9] Ron Minnich. V9fs: A private name space system for unix and its uses for distributed and cluster computing.

[10] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, , and K. Yazawa. The design and implementation of a first-generation cell processor. In *Custom Integrated Circuits Conference*, 2005.

[11] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[12] Gregory R. Watson, Matthew J. Sottile, Ronald G. Minnich, Sung-Eun Choi, and Erik A. Hendriks. Pink: A 1024-node single-system image linux cluster. In *HPCASIA '04: Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference on (HPCAsia'04)*, pages 454–461, Washington, DC, USA, 2004. IEEE Computer Society.